Interaction and observation, categorically

Vincenzo Ciancia*
Institute for Logic, Language and Computation
University of Amsterdam

This paper proposes to use *dialgebras* to specify the semantics of interactive systems in a natural way. Dialgebras are a conservative extension of coalgebras. In this categorical model, from the point of view that we provide, the notions of observation and interaction are separate features. This is useful, for example, in the specification of process equivalences, which are obtained as kernels of the homomorphisms of dialgebras. As an example we present the asynchronous semantics of the CCS.

1 Introduction

The notions of *interaction* and *observation* play a key role in the semantics of concurrent and interactive systems. An *interactive* system or process (imagine a web service, or an operating system) is typically not required to terminate, but it is not always equivalent to the deadlocked machine. This is because, along the execution of a system, the external environment is allowed to interact with the program and observe some side effects (typically, output from the system itself).

However clear in principle, this intuition is lost whenever the semantics of an interactive system is modelled using *labelled transition systems* (LTSs) or their categorical generalisation, the so-called *coalgebras*. The reason is that every interaction that a system makes with the external world, be it originated from the environment, or from an internal action of the system itself, is described in the same way, as a transition from one state to the next.

In this work we turn our attention to a class of categorical models called *dialgebras*. Dialgebras are a straightforward generalisation of both algebras and coalgebras. We interpret these models as a framework where one can describe separately the states of the system, the interactions that the environment and a process may have in each state, and the resulting observations. In our interpretation, dialgebras provide side-effecting operations, therefore providing both contexts and observations simultaneously.

The above is strongly reminiscent of the distinction between input and output in computer science. Thinking of interaction with the environment as an input to a process, and observation as its output, Mealy machines [5] come to mind. These are functions $I \times X \to O \times X$, for X, I and O the set of states of the system, possible input values, and possible output values, respectively. It turns out that one of the simplest and more familiar examples of a dialgebra is a Mealy machine; in the same fashion, one of the simplest and more familiar examples of coalgebra is an LTS. This motivates the following slogan.

Coalgebras generalise labelled transition systems; dialgebras generalise Mealy machines.

As it happens with coalgebras w.r.t. LTSs, the merit of the generalisation is in the fact that, since dialgebras form a category, these generalised Mealy machines are now equipped with a standard notion of equivalence, which is given by the kernel of morphisms of the category.

So, in our framework, the semantics of a programming language is given in terms of a dialgebra. The latter, as we will see, is a function f from a set FX to a set BX. F and B are parametrised in X, which is

*Research supported by the Netherlands Organization for Scientific Research VICI grant 639.073.501

Bliudze, S., Bruni, R., Carbone, M., Silva, A. (Eds.); ICE 2011
EPTCS 59, 2011, pp. 25–36, doi:10.4204/EPTCS.59.3

the set of states of a system. F describes a type of *experiments* that an ideal observer can conduct. Then, results are observed, belonging to the set BX of possible *observations*. The way to define the semantics is by choosing appropriate experiments and observations, and defining such a function f. From this information, using a small amount of category theory, a standard equivalence relation, called dialgebraic bisimilarity, is defined on X. Roughly speaking, two processes are dialgebraic bisimilar if they exhibit the same observations in the same experiments, and the states they reach after the experiments are bisimilar.

An example where it is useful to distinguish between interaction and observation is *asynchronous* semantics. Asynchronous communication may be summarised by saying that "the observer can not see the input actions of a process". More precisely, the observer can not tell input actions from internal computations. In the dialgebraic perspective that we propose on asynchrony, the observer can either sit and look at the system, seeing its output and internal computations, or try to send messages to it. However, a process can either read a message, or consume a message without actually reading it, and store it for later processing. The observer can not tell the two cases apart.

We provide a dialgebraic semantics of the asynchronous CCS, and prove that the obtained equivalence relation coincides with strong asynchronous bisimilarity. In this case, we make a distinction between an underlying *operational semantics* which is expressed by the well-known LTS for the CCS, and the dialgebraic semantics, built on top of it, which specifies the semantic equivalence relation. Bisimilarity of the LTS of the operational semantics, which is also the *synchronous* semantics, is not taken into account in the definition of the dialgebraic semantics.

Using a LTS is not necessary at all to specify a dialgebra. We do so mostly for the sake of simplicity: the asynchronous LTS semantics of process calculi is already well-understood. The operational semantics could in turn be defined as a dialgebra directly on the structure of processes (see §7 for a brief discussion). On the other hand, the usage of a (however specified) operational semantics upon which a process equivalence is based can be considered at least a recurring pattern for the design of process equivalences. The definition of the semantic equivalence may be split in three steps, that we call *execute*, *interact*, *observe*:

execute: the system is run by the means of its operational semantics, specifying some side effects of the process at each state of its execution;

interact: the observer does experiments on the running system;

observe: results are collected, allowing the observer to classify processes by how they react to experiments, giving rise to the behavioural equivalence of choice.

In coalgebras, these three steps are often tied to each other and not so easily separated. Dialgebras give us a different perspective on bisimilarity, where some actions are originated by a running process, and some others by the external environment. The process and the environment may be very different, and the syntax of experiments is not (necessarily) the same as the syntax of processes. This is not so uncommon. Think e.g. of analysis or monitoring for security protocols. The entities (systems) that are being "observed" may be unknown machines or even human beings. The syntax of experiments conducted on such entities may have nothing in common with the entities themselves.

Example 1. For a classical example, think of an human (the observer) in front of a drink-vending machine. The observer can make experiments, such as pressing the buttons, inserting coins etc. A precondition for being able to tell something (and eventually get a drink) is that the machine is running. That is, a current state of the machine is defined, and the machine has an underlying *operational semantics*, which is what the machine really does, independently from what the observer sees. While the

machine is running, the observer performs its experiments, and observes some side-effects. The machines reaches a new state. This is an example where the "syntax of experiments" (e.g. inserting a coin, or pressing a button) is not the "syntax of the vending machine" which would be describing its internal mechanics.

Related work. The study of dialgebras in computer science was initiated in [4] for the categorical specification of data types, and further investigated for the same purpose in [9]. So far, they have not been explored in detail. In this work we divert from the earlier research line: we find applications of dialgebras to programming language semantics, and look at the behavioural equivalences they induce on processes. Moreover, even though we do not provide examples in the current paper, we do not restrict our attention just to the polynomial functors as the syntax of experiments (therefore, we use the equivalences from kernels of morphisms instead of the relational lifting used in [9]). This is since we expect that more complex functors may have useful applications (see §7).

Map of the paper. In §2 we give the definitions of algebras and coalgebras, for comparison with dialgebras. In §3 we give the definition of a dialgebra and explain their intended use. In §4 we present the asynchronous semantics of the CCS. In §5 we give a dialgebraic semantics to the CCS that coincides with the asynchronous one. In §6 we informally discuss other examples of dialgebras. Finally in §7 we sketch some possible future directions.

2 Algebras and Coalgebras

Algebras and coalgebras provide an established methodology for the specification of programming language syntax and semantics. We give here a brief introduction to the definitions of algebra and coalgebra in a category, tailored to a comparison between these two constructions and that of a dialgebra. For more details and pointers to the rich existing literature on algebras and coalgebras, see [10].

First we give the preliminary notion of a *kernel*. For the category-theoretical concepts that we mention, we refer the reader to some basic category theory book (see e.g. [2]).

Definition 1. The kernel of $f : X \to Y$ in a category C is the pullback (if it exists) of the diagram f, f.

When $C = Set$, the kernel of f (up-to isomorphism) is the set $ker\, f = \{(x_1, x_2) \in X \times X \mid f(x_1) = f(x_2)\}$, equipped with the two obvious projections; this is an equivalence relation on X.

Definition 2. *(algebra)* Given a endofunctor F in a category C, an F-algebra is a pair $(X, f : FX \to X)$. An *homomorphism* between two F-algebras (X, f) and (Y, g) is an arrow $h : X \to Y$ such that $h \circ f = g \circ Fh$, that is, the following diagram commutes:

$$\begin{array}{ccc} FX & \xrightarrow{Fh} & FY \\ \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle g} \\ X & \xrightarrow{h} & Y \end{array}$$

When F is a *polynomial* functor, and C is *Set*, then the notion of F-algebra coincides with the classical notion of algebra for a signature (to recover the full power of equational specifications, one needs the stronger notion of algebra of a *monad*, which is out of the scope of this discussion).

Reminder: algebras specify *operations* on the elements of a set.

For example, one can specify the signature (not the equations) of a monoid by providing a set X and the interpretation of composition and identity. In other words, a monoid can be regarded as an algebra for the functor $FX = 1 + X \times X$, that is, a set X and a function $f : 1 + (X \times X) \to X$. The function f is the co-pairing of $f_1 : 1 \to X$, which is the interpretation of the identity of the monoid, and $f_\times : X \times X \to X$, which interprets composition.

Of particular relevance for programming language semantics is that algebras specify the *abstract syntax* of programming languages, by providing operations on abstract syntax terms that can be applied to build larger terms. The functor F provides a syntax to describe operations on elements, and an algebra (X, f) gives the semantics of such a syntax, by computing elements out of these operations.

Definition 3. *(coalgebra)* Given an endofunctor B in a category C, a B-coalgebra is a pair $(X, f : X \to BX)$. An *homomorphism* between two B-coalgebras (X, f) and (Y, g) is an arrow $h : X \to Y$ such that $Bh \circ f = g \circ h$, that is, the following diagram commutes:

$$\begin{array}{ccc} X & \xrightarrow{h} & Y \\ \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle g} \\ BX & \xrightarrow{Bh} & BY \end{array}$$

A coalgebra in the category *Set* of sets and functions is a function $f : X \to BX$ for some behavioural endofunctor $B : X \to X$. The action of B on objects yields a set BX for each X, which is intended to be the *transition type* or *observation type* of the system.

When $BX = \mathscr{P}_{fin}(L \times X)$ and C is *Set*, so that X is a set, then a B-coalgebra f coincides with the classical notion of labelled transition system (LTS) with labels in L. Here, X is the set of states of the system, L is the set of labels, and for all $x \in X$, $f(x)$ is a set of *labelled transitions*, that is, pairs (ℓ, x') consisting of a label and a destination state.

Reminder: coalgebras specify *observations* on the elements of a set.

For example, one can specify an interactive system by providing a set X of *states*, and a transition function $f : X \to \mathscr{P}_{fin}(L \times X)$ describing the non-deterministic observations that we can make about the execution of a process, such as an input, an output, or an internal computation. It is useful to think of L, in this specific case, as the type of *side effects* of the program execution.

The crucial fact about coalgebras is that they form a category, and the natural equivalence relation obtained by the kernel of homomorphisms generalises bisimilarity of LTSs.

By changing the transition type B, one gains generality w.r.t. LTSs. For instance, one can use the probability distribution functor $\mathscr{D}$ in combination with other functors to express various degrees of probabilistic systems [11].

3 Dialgebras

Behavioural equivalences, such as bisimilarity, are typically not based on the syntax of processes. Rather, an external *observer* is assumed, that can see their behaviour. Processes are equivalent when the external observer can not tell them apart.

In this section we introduce dialgebras. We will see that the natural equivalence relation induced by morphisms is still based on behaviours. However, the external observer is now endowed with the power to interact with the system, by doing *experiments* and *observing* the results.

Definition 4. *(dialgebra)* Given a category C, and two endofunctors[1] $F,B : C \to C$, a (F,B)-*dialgebra* is a pair (X, f) where X is an object and $f : FX \to BX$ is an arrow of C.

We will just refer to such a structure as a *dialgebra* when F and B are clear from the context. In the remainder of this section, let us fix two endofunctors F and B.

We call F the *interaction* functor, as it is intended to provide a syntax for constructing experiments. The functor B is the *observation* functor, which is the type of the observed results.

Definition 5. *(dialgebra homomorphism)* Given two dialgebras (X, f) and (Y,g), a *dialgebra homomorphism* from (X, f) to (Y,g) is an arrow $h : X \to Y$ such that $g \circ Fh = Bh \circ f$, that is, the following diagram commutes

$$\begin{array}{ccc} FX & \xrightarrow{Fh} & FY \\ \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle g} \\ BX & \xrightarrow{Bh} & BY \end{array}$$

(F,B)-dialgebras and their homomorphisms form a category. Clearly, when $B = Id$ (the identity functor) one recovers the category of F-algebras, and when $F = Id$ one recovers the category of B-coalgebras. In this work, we only focus on dialgebras in the category *Set* of sets and functions.

Example 2. Non-deterministic Mealy machines are dialgebras for the functors $FX = I \times X$ and $BX = \mathscr{P}_{fin}(O \times X)$, for I the set of input values and O the set of output values.

A dialgebra allows one to specify a set of experiments FX that, when executed trough f, give rise to observations in BX. For a comparison, we mention *bialgebras*. A bialgebra [12] is a pair (f,g) of an algebra $f : FX \to X$ and a coalgebra $g : X \to BX$ having the same underlying set X. The algebra is used to construct elements, the coalgebra to observe them. Every bialgebra is also a dialgebra (the composite $g \circ f : FX \to BX$). Whereas a bialgebra specifies a set equipped with two separate, although possibly nicely interacting, coalgebraic and algebraic operations, a *dialgebra* specifies a set equipped with operations that behave algebraically and coalgebraically at the same time. The interpretation of the "algebraic operations" (the experiments) of a dialgebra does not yield a result, but rather an observation on it. When using dialgebras, just like in algebras, the observer can formally specify a structure (the experiment) that will be executed; just like in coalgebras, the observer interacts with the system in a step-wise fashion: at each state, an experiment can be conducted, yielding observations and possibly subsequent states, on which further experiments are possible.

Reminder: dialgebras specify *operations* on the elements of a set, that yield *observations* as a result.

We now define the underlying equivalence of a dialgebra.

Definition 6. *(dialgebraic bisimilarity)* Given a dialgebra (X, f), dialgebraic bisimilarity is the relation $\approx \subseteq X \times X$ induced by the kernel of any homomorphisms $h : (X, f) \to (Y,g)$ on the underlying set X. That is, we say that $x \approx y \iff \exists (Y,g).\exists h : (X, f) \to (Y,g).h(x) = h(y)$.

[1] In [4], F and B just are required to have the same codomain, not to be endofunctors. The simplified definition we adopt is sufficient for this paper.

In the rest of the paper, we are going to see how to use dialgebras to model asynchrony. An example characterisation of the equivalence induced by morphisms as a back-and-forth condition, as typical in bisimilarity of LTSs, is given in Definition 12 and Theorem 1.

4 The asynchronous CCS

4.1 Syntax and operational semantics

The *calculus of communicating systems* (CCS) [6] is a simple language for studying interactive systems, featuring interleaved parallel composition and synchronization over named channels. In this paper, we use the asynchronous semantics. The definitions we adopt come from the ones for the π-calculus in [1]; we refer the reader to that work for an in-depth study of asynchrony in process calculi.

Let C denote a countable set of *channels*. Define $L_i = C$, $L_o = \{\overline{c} | c \in C\}$, $L_\tau = \{\tau\}$, $L = L_i \cup L_o \cup L_\tau$, the set of *input labels*, *output labels*, *internal labels*, and *labels*, respectively. These labels are observations on a system, representing sending ($\overline{c}$) or receiving (c) an input signal on a channel c, or doing an internal computation step τ.

Definition 7. *(CCS syntax)* The syntax of the asynchronous CCS is defined by the following grammar, where c ranges over a countable set C of *channel names*.

$$P ::= \emptyset \mid \tau.P \mid c.P \mid \overline{c} \mid P \parallel P \mid P + Q$$

We omit the replication and restriction constructs. This is done for ease of explanation as adding them does not affect our proofs. From now on, let X denote the set of agents. In the syntax, $\emptyset$ represents the empty process, that does nothing; $\tau.P$ performs an internal computation step and then behaves as P; $c.P$ waits for an input signal on channel c, and then behaves as P; $\overline{c}$ sends an output signal on channel c; $P_1 \parallel P_2$ is the parallel composition of P_1 and P_2; $P + Q$ denotes non-deterministic choice.

Definition 8. *(CCS operational semantics)* The operational semantics is given in the form of a LTS $t : X \to \mathscr{P}_{fin}(L \times X)$, defined by the following rules:

$$c.P \xrightarrow{c} P\,(in) \qquad \tau.P \xrightarrow{\tau} P\,(tau) \qquad \overline{c} \xrightarrow{\overline{c}} \emptyset\,(out)$$

$$\frac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q}\,(par) \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \parallel Q \xrightarrow{\alpha} P \parallel Q'}\,(par') \qquad \frac{P \xrightarrow{c} P' \quad Q \xrightarrow{\overline{c}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}\,(syn)$$

$$\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}\,(sum) \qquad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}\,(sum')$$

Rules (in), (tau), and (out) are straightforward. Rules (par) and (par') allow components to run in parallel in an interleaved fashion. Rule (syn) allows a process that can do an input and a process that can do an output to synchronise. Rules (sum) and (sum') allow a non-deterministic choice to take place.

4.2 Asynchronous bisimilarity

We define asynchronous bisimulation and bisimilarity directly for CCS terms.

Definition 9. *(CCS asynchronous bisimilarity)* A relation $R \subseteq X \times X$ is an *asynchronous simulation* if and only if, whenever $(x,y) \in R$, and $x \xrightarrow{\alpha} x'$, then there is y' such that:

- if $\alpha = \tau$ or $\alpha = \overline{c}$ for some c, then $y \xrightarrow{\alpha} y'$ and $(x',y') \in R$;
- if $\alpha = c$ for some c, then $\overline{c} \parallel y \xrightarrow{\tau} y'$ and $(x',y') \in R$
  or, equivalently
  if $\alpha = c$ for some c, then $(x',y') \in R$ and either $y \xrightarrow{c} y'$ or $y \xrightarrow{\tau} y''$ with $y' = \overline{c} \parallel y''$.

An *asynchronous bisimulation* is a simulation R such that R^{-1} is a simulation. *Asynchronous bisimilarity* is the largest bisimulation.

We write $x \sim y$ whenever x is asynchronous bisimilar to y, or equivalently there is some asynchronous bisimulation R such that $(x,y) \in R$. In asynchronous bisimilarity, input labels can be matched "loosely" by a τ transition that stores an output process in parallel with the execution. We are going to see how to turn this definition into dialgebraic bisimilarity. Before that, we remark that synchronous bisimilarity (that would be obtained by employing strong bisimilarity on the LTS from Definition 8) is included in the asynchronous one. The inclusion is strict. Two processes that are not synchronous bisimilar but are asynchronous bisimilar are $c.\overline{c}.\emptyset + \tau.\emptyset$ and $\tau.\emptyset$ (example adapted from [1], where a thorough discussion can be found).

5 Observing interactions

Asynchronous bisimilarity does not coincide with the coalgebraic bisimilarity obtained from the transition system of Definition 8. We define a dialgebra whose set of states is that of the CCS agents, and where dialgebraic bisimilarity is asynchronous bisimilarity.

5.1 Dialgebra for the asynchronous CCS

First, we define, and fix hereafter, a specific pair of interaction and observation functors.

Definition 10. *(CCS interaction and observation functors)* We let the interaction functor be $FX = X + L_o \times X$, and the observation functor be $BX = \mathscr{P}_{fin}((L_o \cup L_\tau) \times X)$.

For any set X, an element e of the disjoint union FX is either in the form x or $(\overline{c},x)$, for $c \in C$ and $x \in X$. Roughly, e is the syntax of an experiment where we can either observe the execution of x, or send a signal to x on channel c. An element t of BX is a set of pairs $(\overline{c},x')$ or (τ,x') for $c \in C$ and $x' \in X$. The element t is a transition to x' labelled with either the observation of an output signal on a certain channel, or of an internal computation step. No input labels appear. Input is modelled as the argument of a function, instead of as a side-effect. This is in line with the idea that input is an action of the environment, not an action of the process.

We now define a (F,B)-dialgebra for the CCS. From now on, whenever f is a dialgebra, we use the shorthand $e \xrightarrow{\beta}_f x'$ to denote that $(\beta,x') \in f(e)$, and omit f when clear from the context.

Definition 11. *(CCS dialgebraic semantics)* The (F,B)-dialgebra $f : FX \to BX$, where X is the set of CCS processes equipped with the operational semantics of Definition 8, is defined by the following rules:

$$\frac{x \xrightarrow{\alpha} x' \quad \alpha = \tau \vee \alpha = \overline{c}}{x \xrightarrow{\alpha}_f x'}\,(run) \qquad \frac{x \xrightarrow{c} x'}{(\overline{c},x) \xrightarrow{\tau}_f x'}\,(in) \qquad \frac{x \xrightarrow{\tau} x'}{(\overline{c},x) \xrightarrow{\tau}_f \overline{c} \parallel x'}\,(store)$$

Premises of rules use the operational semantics of Definition 8. Rule (run) expresses the fact that we can observe the output and internal computation steps of a system. Rule (in) states that whenever a process x can do input, the experiment $(\overline{c},x)$ yields the observation of an internal computation step. By Rule $(store)$, whenever a process can do an internal computation step, then it can also store an input signal from the environment for subsequent processing. The observations for the (in) and $(store)$ rules are the same, therefore an observer can not distinguish the application of either one of the two rules.

5.2 Characterising dialgebraic bisimilarity

A characterization of the equivalence induced by dialgebra homomorphisms for the functors F and B of Definition 10 can be given as follows.

Definition 12. *(Back-and-forth bisimilarity of dialgebras)* Given a (F,B)-dialgebra $f : FX \to BX$, a relation $R \subseteq X \times X$ is a *back-and-forth simulation* if and only if, for all $(x,y) \in R$ and $c \in C$:

1. whenever $x \xrightarrow{\alpha}_f x'$, there is y' such that $y \xrightarrow{\alpha}_f y'$ and $(x',y') \in R$;
2. whenever $(\overline{c},x) \xrightarrow{\tau}_f x'$, there is y' such that $(\overline{c},y) \xrightarrow{\tau}_f y'$ and $(x',y') \in R$.

A bisimulation is a simulation R such that R^{-1} is a simulation. Two elements of X are said *bisimilar* if and only if there is a bisimulation relating them. The corresponding relation is called bisimilarity.

We write $x \simeq y$ to denote that x is bisimilar to y.

Proposition 1. *Back-and-forth bisimilarity is an equivalence relation.*

Theorem 1. (back-and-forth vs. kernel) *When F and B are as in Definition 10, dialgebraic bisimilarity from Definition 6 and back-and-forth bisimilarity from Definition 12 coincide.*

Proof. Fix a dialgebra (X,f). First, consider a dialgebra (Y,g) and $h : (X,f) \to (Y,g)$. We show that $\ker h$ is a back-and-forth bisimulation, therefore it is included in $\simeq$. Assume $hx = hy$ for some $x,y \in X$. For all $\alpha \in L$, by definition of homomorphism, we have $g(Fh(\alpha,x)) = Bh(f(\alpha,x))$. Therefore $g(\alpha,hy) = Bh(f(\alpha,x))$. Let $(\beta,x') \in f(\alpha,x)$. Then $(\beta,hx') \in Bh(f(\alpha,x))$, therefore $(\beta,hx') \in g(\alpha,hy) = g(Fh(\alpha,y))$, thus by commutativity $(\beta,hx') \in Bh(f(\alpha,y))$. Then there is some y' such that $(\beta,y') \in f(\alpha,y)$ and $hx' = hy'$. This proves that $\ker h$ is a simulation. Notice that the kernel of a function is an equivalence relation, therefore $(\ker h)^{-1} = \ker h$), thus proving that $\ker h$ is a bisimulation. For the other direction of the proof, let $[x]$ denote the equivalence class of x in $X_{/\simeq}$. Consider the quotient dialgebra $(X_{/\simeq}, f_{/\simeq})$, with $f_{/\simeq}(\alpha,[x]) = \{(\beta,[x'])|(\beta,x') \in f(x)\}$. Notice that $f_{/\simeq}$ is well defined by definition of $\simeq$. The quotient function $hx = [x]$ is obviously a homomorphism of dialgebras, and it is the case that whenever $x \simeq y$ then $h(x) = h(y)$. □

Finally, we prove that asynchronous and back-and-forth bisimilarity coincide.

Theorem 2. (asynchronous vs. back-and-forth) *Asynchronous bisimilarity from Definition 9 and back-and-forth bisimilarity coincide for the set X of CCS agents, that is: for all $x,y \in X$, we have $x \sim y$ if and only if $x \simeq y$. Therefore, by Theorem 1, asynchronous bisimilarity and dialgebraic bisimilarity coincide.*

Proof. We provide the proof just for completeness, as it is immediate from the characterisation of asynchronous bisimilarity as a 1-bisimilarity in [1]. We prove that $\sim$ is a back-and-forth bisimulation. Symmetry, and Case 1 from Definition 12 are obvious. For Case 2, suppose $(\overline{c},x) \xrightarrow{\tau} x'$. Then we distinguish two cases.

- if Rule (in) is applied to $(\bar{c},x)$, we have $x \xrightarrow{c} x'$. We now look at Definition 9. Since $x \sim y$, we have $\bar{c} \parallel y \xrightarrow{\tau} y'$ with $x' \sim y'$. We inspect the rules in Definition 8. The rules that can be applied to $\bar{c} \parallel y$ are (par) and (syn) (and (par') which is treated in the same way as (par)). Therefore we have either $y \xrightarrow{\tau} y''$ with $y' = \bar{c} \parallel y''$, or $y \xrightarrow{c} y'$. By applying either Rule (in) or $(store)$ from Definition 11, we obtain $(\bar{c},y) \xrightarrow{\tau}_f y'$ and since $x' \sim y'$ we get the thesis.
- if Rule $(store)$ is applied to $(\bar{c},x)$, then $x \xrightarrow{\tau} x''$ with $x' = \bar{c} \parallel x''$. Therefore, $y \xrightarrow{\tau} y''$ and $x'' \sim y''$. It is well known and easy to prove that $x'' \sim y'' \implies \bar{c} \parallel x'' \sim \bar{c} \parallel y''$. Therefore by applying Rule $(store)$ we get $(\bar{c},y) \xrightarrow{\tau} y'$ and $x' \sim y'$, q.e.d.

Next, we prove that $\simeq$ is an asynchronous bisimulation. Suppose $x \simeq y$ and $x \xrightarrow{\alpha} x'$. We look at Definition 9. The cases for $\alpha = \tau$ or $\alpha = \bar{c}$ are obvious. Suppose $\alpha = c$ for some c. By Rule (in) in Definition 11 we have $(\bar{c},x) \xrightarrow{\tau}_f x'$ and by $x \simeq y$ we get $(\bar{c},y) \xrightarrow{\tau}_f y'$ with $x' \simeq y'$. Either Rule (in) or $(store)$ from Definition 11 can be applied to $(\bar{c},y)$. Therefore either $y \xrightarrow{c} y'$, or $y \xrightarrow{\tau} y''$ with $y' = \bar{c} \parallel y''$. In both cases, we have $\bar{c} \parallel y \xrightarrow{\tau} y'$ and $x' \simeq y'$, from which the thesis. □

6 Discussion on further examples

The example that we present is very simple, and purposed to illustrate just the idea of an observer that can interact with the examined system. More interesting dialgebras can be described by either moving to a richer category than *Set*, or by changing the interaction and observation functor. We briefly describe some possible constructions, whose detailed study is left for future work.

Complex systems Consider dialgebras of the form $f : \mathscr{P}_{fin}(X) \to L \times \mathscr{P}_{fin}(X)$. At each step in time, from a set $p \in \mathscr{P}_{fin}(X)$, a side effect in L is observed, and a new set of elements p' is obtained. Such a function may be used to represent systems where the semantics depends on a number of entities that collaborate. At each step in time, the system evolves, some old elements may be "destroyed" and new elements can be created, while some side effect in L takes place. The behaviour of the system is *more than the sum of its parts*, in the sense that it is not determined by the behaviour of singletons. The semantics of $\{x\}$, that is, x in isolation, may be totally unrelated to the semantics of, say, the set $\{x,y\}$. Notice that $f : \mathscr{P}_{fin}(X) \to L \times \mathscr{P}_{fin}(X)$ is also a coalgebra in *Set* for the functor $T(X) = L \times X$, having $\mathscr{P}_{fin}(X)$ as underlying set. However, it's obvious that the obtained notion of bisimulation is not the same, even by just looking at types. Seeing f as a coalgebra, one gets a relation on $\mathscr{P}_{fin}(X)$; seeing it as a dialgebra, one gets a relation on X, that takes into account how elements behave when joined to the same sets of other elements.

Chemical reactions In many cases programming language semantics has been inspired by chemical and biological processes. Consider the finite multi-set functor $\mathscr{M}(X) = \{m : X \to \mathbb{N} \mid \{x \mid m(x) \neq 0\} \text{ is finite}\}$. Think of X as a set of elements that take part in *reactions* in variable quantities. A dialgebra $f : \mathscr{M}(X) \to \mathscr{M}(X)$ specifies how a given reaction evolves by creating a multi-set of products from a multi-set of reagents. The obtained notion of bisimilarity makes reagents equivalent when substituting one with the other in any reaction yields equivalent products, in the same quantities.

The π-calculus A very similar development to the one presented here, exemplifying the use of a different base category, is the semantics of the asynchronous π-calculus. Similarly to what happens for

the synchronous pi-calculus and coalgebras [3], one would use the functor category Set^I, where I is the category of of finite sets and injections. The semantics would involve the endofunctor for fresh name allocation δ which is typical of functor categories, which is needed to properly model bound output. Dialgebras using δ correspond to Mealy machines with name allocation along output, whose study is possibly of interest independently from the specific application of the π-calculus.

Testing semantics Even though we spoke of interaction and observation, we did not mention so far the family of *testing equivalences* (see [7]), where interaction and observation play a key role. Testing equivalences are defined as those obtained by putting a process in parallel with an arbitrary other process making use of a distinguished channel. Output on such channel signals that a test has been successful. Binary dialgebras come to mind as an effective way to represent such kinds of equivalence relations. However, in testing equivalences, one is not able to observe how many synchronisation steps between processes are needed before the success signal is sent. Such a semantics could be defined by observing the behaviour of a process as a single "big step"; however, this would defeat the implicit coinductive properties of dialgebras. A common feature of dialgebras and coalgebras is that observations lead to successor states, and then in a coinductive fashion further experiments/observations can be done on these successor states. However, in the case of testing equivalences, there is no successor state: once success is signalled, the experiment is concluded. Further investigation may yield non-obvious coinductive ways to represent these kind of relations on processes.

7 Conclusions and future work

The construction we have seen in §5 has obvious similarities with barbed equivalence and with the asynchronous semantics of the π-calculus by Honda and Tokoro (both described in [1]). That's expectable, since in the end we are trying to describe the same equivalence relation.

In the case of the asynchronous CCS, it is not difficult to recover a coalgebraic semantics. This is done by translating the dialgebraic semantics along the isomorphisms $X + L_o \times X \to \mathscr{P}_{fin}((L_o + L_\tau) \times X) \cong (L_i + 1) \times X \to \mathscr{P}_{fin}((L_o + L_\tau) \times X) \cong X \to (\mathscr{P}_{fin}((L_o + L_\tau) \times X))^{L_i+1}$ (indeed, after noting that $L_i \cong L_o$). Notice that the latter is genuinely a coalgebra for the functor $(\mathscr{P}_{fin}(L_o + L_\tau) \times -)^{L_i+1}$. It is not difficult to see that such a translation preserves and reflects the equivalence induced by kernels of homomorphisms (of dialgebras in one case, of coalgebras in the other).

Even though it might be interesting to derive a coalgebraic semantics for the asynchronous CCS, we do not discuss the details of such a construction: the purpose of using this language as an example is not to provide a new semantics for asynchronous process calculi. Rather, the asynchronous CCS is possibly the simplest language where it makes sense to distinguish between moves of the environment and moves of the system being examined in order to define the semantics. Our aim is to show how such a distinction is naturally encoded using dialgebras, and their built-in definition of behavioural equivalence makes them appealing as an alternative to coalgebras in the specification of interactive systems.

We summarise below some possible future directions and open questions.

Inductively defined dialgebras. We defined a dialgebra for the asynchronous CCS by assuming an existing operational semantics. It is indeed possible to specify such a semantics using dialgebras. First, because coalgebras actually *are* dialgebras with $F = Id$. Moreover, one could easily define an (F, B)-dialgebra, for F and B as in §5, directly by induction on terms forming the set of agents X, in the same fashion of bialgebras and distributive laws. It would be relevant to study distributive laws and

specification languages for inductively defined dialgebras, following the same route of bialgebras. Doing so, it would be possible to guarantee that a given dialgebraic semantics of a calculus is also a congruence with respect to the operators of the algebra describing its syntax.

Logics Dialgebras are equipped in [9] with *dialgebraic specifications*, even though neither a full adequacy result relating logical equivalence and bisimilarity, nor Birkoff-style theorems are established. It ought to be clarified what is a logical formalism that adequately specifies dialgebras. Such a logic would be an intermediate language between modal and equational logic. The work [8], relating dialgebras to the so-called *abstract logics* is possibly relevant. This research line should take advantage of, and extend, the many existing studies in the field of coalgebraic modal logic.

Non-polynomial interaction functors Dialgebras are parametrised in the interaction and observation functors. Non-polynomial interaction functors, such as e.g. a probability distribution over the input values, could provide valuable case studies. Modulo the observation functor being "probabilised", too, such dialgebras may be used to represent a kind of probabilistic Mealy machines, where the probability distribution of the input determines that of the output. It should be understood whether in the case of non-polynomial interaction functors there is some gain in expressive power w.r.t. coalgebras.

Minimisation Coalgebras have an elegant and simple minimisation procedure, based on *iteration along the terminal sequence* and generalising partition refinement for automata. Are there canonical models in dialgebras? The results in [9] seem to point out that such a theory would be very difficult in the presence of so-called *binary methods*, due to non-closure of bisimulations under union, and the lack of a final dialgebra. However, the (dialgebraic) bisimilarity quotient may still exist in interesting cases. More work is required on this side. The precise conditions when final dialgebras and bisimilarity quotients exist should be clarified. Also notice that in [9] F is assumed to be polynomial. Since we seek for non-polynomial interaction functors too, we expect that some work on the side of canonical models will be needed in order to understand how bisimilarity of dialgebra can be decided, possibly by finite representations derived from the definitions of F and B.

References

[1] Roberto M. Amadio, Ilaria Castellani & Davide Sangiorgi (1998): *On bisimulations for the asynchronous pi-calculus*. *Theoretical Computer Science* 195(2), pp. 291 – 324. doi:10.1007/3-540-61604-7_53

[2] Steve Awodey (2010): *Category Theory (Oxford Logic Guides)*, 2 edition. Oxford University Press, USA.doi:10.1093/acprof:oso/9780198568612.001.0001

[3] Marcelo P. Fiore & Daniele Turi (2001): *Semantics of Name and Value Passing*. In: *16th Annual IEEE Symposium on Logic in Computer Science (LICS)*, IEEE Computer Society, pp. 93–104. doi:10.1109/LICS.2001.932486

[4] Tatsuya Hagino (1987): *A Categorical Programming Language*. Ph.D. thesis, University of Edinburgh.

[5] G.H. Mealy (1955): *A Method to Synthesizing Sequential Circuits*. *Bell System Technical Journal* , pp. 1045–1079.

[6] R. Milner (1982): *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc.

[7] Rocco De Nicola & Matthew Hennessy (1984): *Testing Equivalences for Processes*. *Theoretical Computer Science* 34, pp. 83–133. doi:10.1016/0304-3975(84)90113-0

[8] Alessandra Palmigiano (2002): *Abstract Logics as Dialgebras*. *Electronic Notes in Theoretical Computer Science* 65(1), pp. 254–269. CMCS'2002, Coalgebraic Methods in Computer Science. doi:10.1016/S1571-0661(04)80367-0

[9] Erik Poll & Jan Zwanenburg (2001): *From Algebras and Coalgebras to Dialgebras*. *Electronic Notes in Theoretical Computer Science* 44(1), pp. 289 – 307. CMCS'2001, Coalgebraic Methods in Computer Science. doi:10.1016/S1571-0661(04)80915-0

[10] J. J. M. M. Rutten (2000): *Universal coalgebra: a theory of systems*. *Theoretical Computer Science* 249(1), pp. 3 – 80. doi:10.1016/S0304-3975(00)00056-6

[11] A. Sokolova (2005): *Coalgebraic Analysis of Probabilistic Systems*. Ph.D. thesis, TU Eindhoven.

[12] Daniele Turi & Gordon Plotkin (1997): *Towards a Mathematical Operational Semantics*. In: *12th Annual IEEE Symposium on Logic in Computer Science (LICS)*, IEEE Computer Society, pp. 280–291. doi:10.1109/LICS.1997.614955